Style Guide

Programming Foundations II

For each programming assignment, 10 points (of the 100 total) come from proper use of style in your source code. Any student wishing to get full credit for an assignment will therefore have to strictly adhere to these guidelines. Style guides like these are common in industrial settings, so learning to follow one is an important skill to master if you want to eventually pursue a career in software development.

1. Legibility

Indentation: All source code should be properly indented. Consecutive statements within a block should all have the same level of indentation, and sub-blocks should be indented one additional level. Be careful NOT to combine tabs and spaces. The tab size varies depending on the text editor, so using it for alignment will often render your code unreadable when someone else looks at your code. Many editors support the ability to insert a certain number of spaces (e.g. 4) (instead of a tab character) whenever you press the tab key. If you enable this setting, you will rarely have indentation issues with your code.

Braces: Braces should be consistent as well. In C++, it is common practice to align braces vertically (balanced braces, see the good example below), rather than leaving them misaligned (broken braces). Generally, you should adhere to the general conventions of whichever language you're working in. In Java, for example, it is more common to see broken braces than balanced ones.

Whitespace: Proper use of whitespace is also important in order to maintain readability (both in the source code as well as any output generated by your program). You should include blank lines between function implementations and between important segments in functions. Also include spaces where it will help to improve legibility. It is generally wise to leave one space before and after an operator, for example.

Below are examples of bad legibility and good legibility for a simple averaging function.

Bad

```
double average(int data[5])
{    int sum=0;
  for (int i=0; i<5; ++i)
    {
      sum+=data[i];}
    return ((double)sum
/ 5.0);}
```

Good

```
double average(int data[5])
{
    int sum = 0;
    for (int i = 0; i < 5; ++i)
    {
        sum += data[i];
    }

    return ((double) sum / 5.0);
}
```

Style Guide

Programming Foundations II

2. Capitalization

Standard naming and capitalization schemes are particularly common in industrial style guides. The convention we will would like you to follow is to use camelCase for all **variables** and **functions** and CapitalCamelCase for all **classes**. Notice that all letters are lowercase except for the beginning of consecutive words. **Constants**, however, are normally written using ALL_CAPS, using underscores to separate words.

These capitalization rules are not always applicable. If your variable represents an acronym (like GPA or NASA), it is common to leave it capitalized since people are used to seeing it used in that manner.

Files that contain classes (e.g. header files and source files) should have names that match the class defined within them (including capitalization). For example, a "Book" class would be defined in "Book.h" and "Book.cpp", rather than "book.h" and "book.cpp".

The table below shows several good and bad examples of capitalization.

Bad	Good

`class student`
`{`
`public:`
`  student();`
`  void Print();`
`  string Get_Name();`
 	
`private:`
`  string Name;`
`  float gPA;`
`  int Age;`
`  int num_Cats;`
`};`
	`class Student`
	`{`
	`public:`
	`  Student();`
	`  void print();`
	`  string getName();`
	
	`private:`
	`  string name;`
	`  float GPA;`
	`  int age;`
	`  int numCats;`
	`};`

Style Guide

Programming Foundations II

3. Comments

Comments are very important for code readability. In order for someone else to understand your code, you should include helpful comments where appropriate. In general, each class should have a comment describing what the purpose of the class is. Functions should (at a minimum) include descriptions of what the various inputs are and what the expected return value is, as well as what it means. If a function's name is not self-descriptive (e.g. getValue()), it is also helpful to include a description of what that function is supposed to do. Lastly, any code segments that are not intuitive should be commented to describe the intended operation.

Note: Try to avoid writing comments in line with the code you're documenting. It's much easier to read your code if comments are written on the line **ABOVE** the code you are documenting.

Note 2: Do not fall into the trap of commenting every single line of code. You should use one comment for each "block", where a block is typically less than 10-20 lines or so.

The table below demonstrates examples of helpful comments.

Style Guide

Programming Foundations II

Class

```
/**
 * This class implements a Linked List of Book objects. The links
 * in the list are BookNode objects. The class also includes
 * several modification methods, as well as a print method.
 */
class BookList
{
    // …
};
```

Function

```
/**
 * Calculates the nth Fibonacci number.
 *
 * @param N. The value of N. fiboonacci(1) = 1, fiboonacci (2) =
 *           1, fiboonacci (3) = 2, etc.
 * @return The Nth Fibonacci number.
 */
int fibonacci(int N)
{
    // …
}
```

General

```
int search(int array[], int min, int max, int number)
{
    // Check if value is not in array
    if (min > max) return -1;

    int med = (min + max) / 2;

    // We've found the answer
    if (array[med] == number)
            return mid;

    // Search the left half of the array
    else if (array[med] > number)
        return search(array, min, med - 1, number);

    // Search the right half of the array
    else if (array[med] < number)
        return search(array, med + 1, max, number);
}
```

Style Guide

Programming Foundations II

4. Program Structure

As you begin to write larger software systems, structuring your programs becomes increasingly more important. Here are some general guidelines that will help with organization. Large components (like lists or other data structures) should be abstracted as classes (with each implemented in its own header and implementation files). Smaller components should be abstracted as general purpose functions in order to maximize reusability. Lastly, make sure each component (such as a class or a function) has exactly one purpose whenever possible. If you have a single function that contains all of the driver logic, for example, that function should be broken up into smaller functions, each of which has a single role. Refactoring (or splitting a large component into smaller, more manageable pieces) will make testing easier and the implementation easier to understand overall.

Hint: Avoid doing any sort of I/O (e.g. cin/cout) inside a function whenever possible. Interfaces for functions should assume that the user can provide all necessary inputs and that the function will return its result (e.g. directly or by reference).

5. Names

It will help anyone reading your code immensely if your names for variables, functions, and classes are chosen carefully. The name should concisely reflect the purpose of the entity. Bad names can be too short, be too long, or poorly describe what the intended purpose is. One exception is loop counters, for which it is common to use single letters, like i, j, and k for variable names. Here are some examples of good and bad names:

Bad	Good
foo	counter
thisVariableRepresentsTheSumOfTheNumbers	sum
v1	numPlayers

Member / Static / Global variables: Many style guides give more specific naming guidelines. In Java, for example, member variables (or class variables) are typically prefaced with the letter “m”. For example, if a class contains an array called “data”, it would be named “mData” instead. A similar approach is used for static variables (prefaced with “s”) and global variables (prefaced with “g”). Such schemes are exceptionally helpful for quickly understanding whether a variable is declared locally (with no prefix), as part of a class (prefix of “m”), statically (prefix of “s”), or globally (prefix of “g”). We will not require you to adhere to this naming scheme, but it is highly recommended, as it is a good practice and tends to improve the code you write significantly.